# Wykład 2

Instrukcje - wprowadzenie

# Cel

- Zilustrować sposób postępowania przy układaniu (projektowaniu) algorytmu.

# Przykład algorytmizacji zadania (metoda intuicyjna)

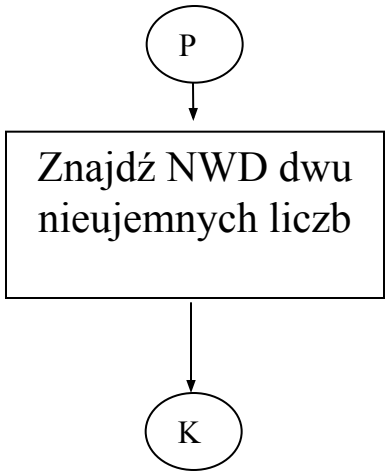
- **Zadanie:**
  - **Dane są liczby całkowite a i b. Znajdź ich największy wspólny podzielnik.**
  - Przykład
    - NWD(21,7)=7
    - NWD(16,6)=2

# Przykład

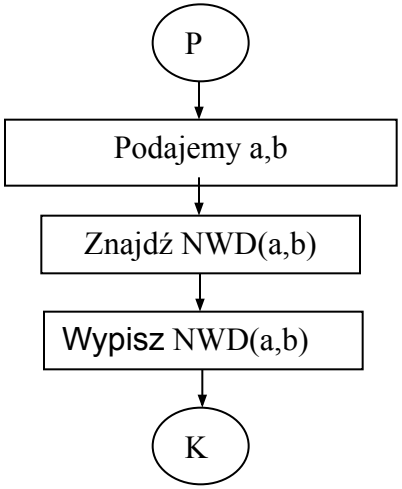
- **Krok I (trywialny) – sformułowanie problemu:**
  - Co jest dane?
  - Odp. Dane są dwie liczby. W razie potrzeby możemy je oznaczyć jako a i b (co nam to daje?).
  - Co to jest największy wspólny podzielnik.
  - Odp. Jest to największa liczba całkowita, przez którą dzielą się bez reszty obydwie liczby. NWD zawsze istnieje (liczba 1).

# Przykład - *Ogólne sformułowanie problemu znalezienia NWD*

- Graficznie można problem przedstawić następująco:



# Przykład - *Dokładniejsze (graficzne) sformułowanie problemu znalezienia NWD*



# Przykład - poszukiwania

- **Krok II (najtrudniejszy)**: Jak obliczyć NWD.
  - Pomysły luźne (burza mózgów) - Pomysły mogą być różne, np. obydwie liczby dzielimy przez liczby od 1 do a i od 1 do b. Badamy, czy liczby, przez które dzielimy dzielą obydwie liczby bez reszty, jeśli tak to zapisujemy na „boku” ich wartość, a potem znajdujemy maksymalną liczbę – sposób kosztowny.
  - Poprawę efektywności można uzyskać zauważając, że wystarczy znaleźć minimalną liczbę z a i b i dzielić przez wszystkie liczby do tej liczby (oczywiste, bo nie jest możliwe by była liczba całkowita np. dla a/(a+1)).
  - Te rozważania pozwalają nam zapoznać się z problemem ale nie jest to systematyczna metoda (o takie metody jest jednak trudno, można mówić jedynie o metodach dla pewnych klas zadań).

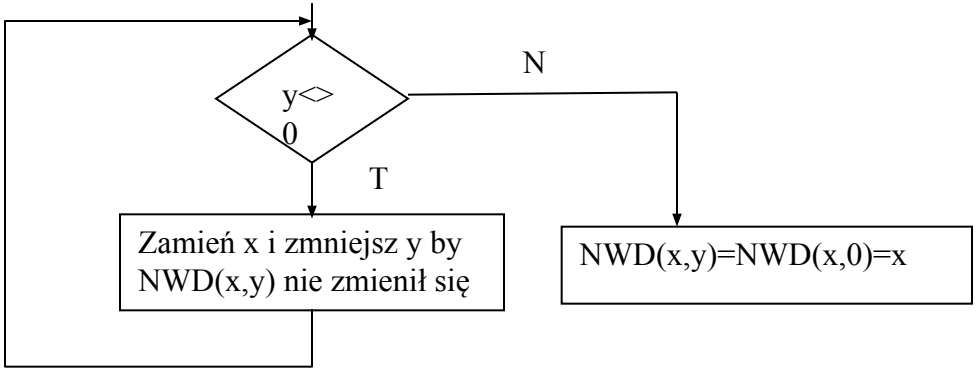
# Przykład - studia

- Pomysł bardziej systematyczny: Najlepsze metody uzyskuje się korzystając z własności tego co chcemy policzyć. Zauważmy, że można pominąć liczby ujemne, bo
  - **nwd(-a,b)=nwd(a,b), dla a,b>0**
  - jest to oczywiste i wynika z definicji nwd.
- Dalej, można zauważyć, że
  - **nwd(a,0)=a,**
- jak również, że
  - **nwd(a,b)=nwd(b,a)**
- co jest też oczywiste.

# Przykład - wnioski

- Gdyby więc udało się nam zmniejszać liczby, dla których szukamy nwd tak, by nie zmieniła się liczba NWD to w końcu doprowadzilibyśmy do sytuacji, że szukalibyśmy NWD(a’,0) lub NWD(0,b’). Intuicyjny algorytm można więc zapisać następująco:
  - **(1) Pod x podstaw a a pod y podstaw b,**
  - **(2) Jeśli y<>0 to**
    - **zmniejsz y i zmień x w ten sposób, by obie liczby pozostały >= 0 i by wartość NWD nie zmieniła się,**
    - **powtórz krok (2),**
  - **(3) jeśli y=0 to NWD(x,0)=x.**

# Przykład - graficznie



# Przykład

- Wszystkie czynności poza krokiem (a) są proste. Ale zauważmy, że (to jest nasza wiedza, doświadczenie, intuicja,...), że
- **x=(x div y)*y+x mod y, (*)**
- gdzie x div y oznacza dzielenie całkowite, x mod y oznacza resztę z dzielenia (takie operatory występują w j.Pascal).
- Np.
  - dla x=30 y=7 mamy:
    - 30=(30 div 7)*7+30 mod 7=4*7+2=30
  - a dla x=7 i y=30 mamy
    - 7=(7 div 30)*30+7 mod 30=0*30+7=7.
- Z (*) mamy:
- **x mod y = x-(x div y)*y. (**)**
- Ten zapis oznacza, że jeśli NWD dzieli x i y to NWD dzieli (wynik będzie liczbą całkowitą) również x-(x div y)*y, bo
- **(x-(x div y)*y)/NWD=x/NWD-(x div y)*(y/NWD)**
- jest na pewno liczbą całkowitą. Zachodzi więc (na podstawie **):
- **NWD(x,y)=NWD(y,x mod y)**

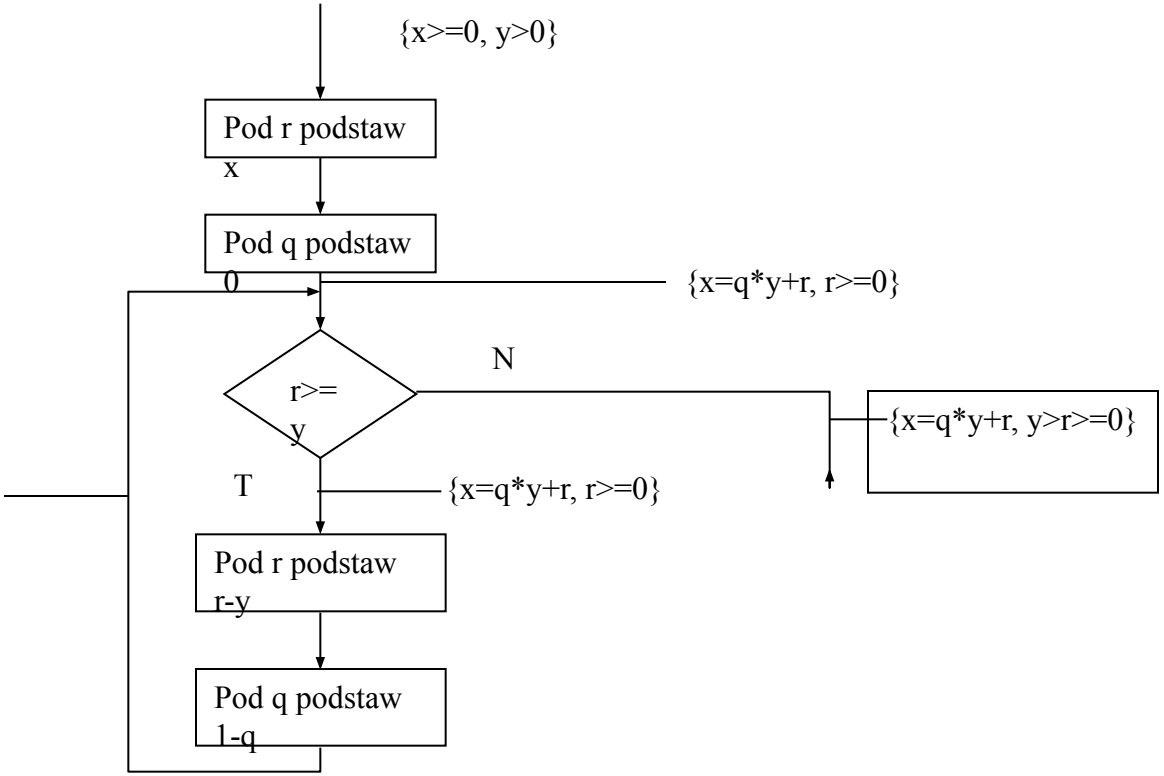
# Przykład – krok a

- Otrzymaliśmy więc sposób na wykonanie kroku (a). Jest on postaci:
- **(a1) Pod r podstaw x mod y,**
- **(a2) Pod x podstaw y,**
- **(a3) Pod y podstaw r.**

# Przykład – problem z mod

- Jak obliczyć x mod y. Należy zauważyć, że gdybyśmy dysponowali umiejętnością obliczania x mod y to byłby to już koniec problemu. Tak nie jest, więc nowy problem to: obliczyć x mod y. Można zauważyć, że obliczanie x mod y jest równoważne znalezieniu takiego q i r, że zachodzi: x=q*y+r, gdzie wielkość r jest szukaną liczbą, ma ona następującą własność:
  - **0<=r<y.**
- Stąd mamy, że
  - **r=x-q*y.**
- Tak więc należy od x odjąć r i sprawdzić, czy wynik spełnia warunek 0<=r<y, jeśli tak, to mamy resztę, jeśli nie to powtarzamy odejmowanie. Tak więc krok (a1) można rozpisać jako:
  - **(a11) Pod q podstaw 0,**
  - **(a12) Pod r podstaw x,**
  - **(a13) Sprawdź, czy 0<=r<y**
  - **(a14) jeśli tak, to została znaleziona reszta,**
  - **(a15) jeśli nie, to**
  - **(a151) Pod r podstaw r-x,**
  - **(a152) Pod q podstaw 1+q**
  - **(a153) Wykonaj algorytm od kroku (a13).**

# Przykład – graficznie obliczanie mod



# Przykład – zakończenie

- Ponieważ wszystkie operacje są już wykonywane za pomocą elementarnych operacji (każda maszyna cyfrowa powinna je mieć) to zadanie jest już rozwiązane.
- Rozwiązanie należy zapisać w języku programowania. W tym momencie pojawia się problem, czy otrzymany program jest dobry.

# Inne zadanie – trochę inne spojrzenie na problemy algorytmiczne (mniej sformalizowane)

Dany jest ciąg n-elementowy liczb całkowitych. Napisz program znajdujący liczbę w tym ciągu o maksymalnej sumie cyfr.

# Szkic rozwiązania:

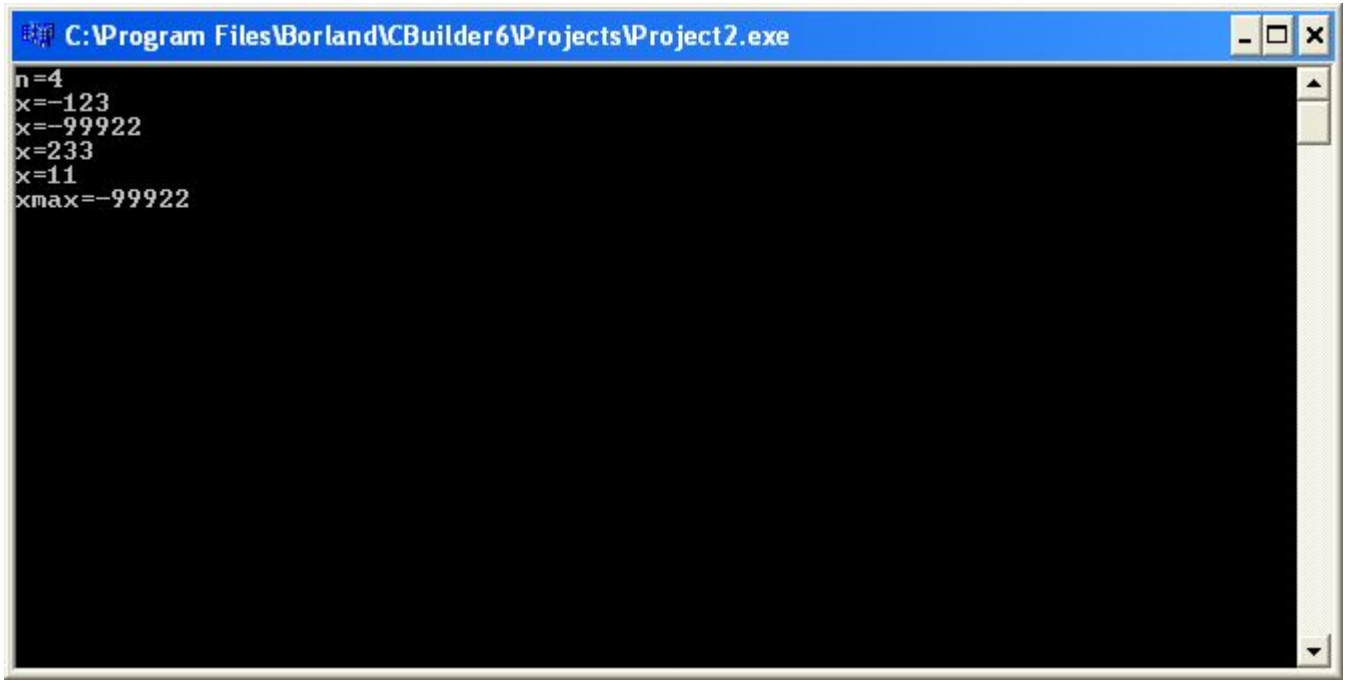
- Wstępne rozważania:
  - Co mamy dane, co poszukujemy. Częsty błąd, to niezrozumienie jak wyglądają dane, np. można wpaść na „genialny” pomysł, że dane to „-23, 45, 33, -23, 321” - to jest tekst, który przetwarza się dużo trudniej, poza tym, nie musimy mieć od razu wszystkich danych, tylko mogą kolejno napływać (np. aplikacje sieciowe, internetowe, programowanie równoległe). Czym jest cyfra, to wiadomo (nie ma cyfr ujemnych!!!!!)
  - Wczytujemy n,
  - Ustawiamy wartości początkowe i, max – i oznacza którą liczbę przetwarzamy, max powinno być takie, by pierwsza liczba je poprawiła,
  - Dla kolejnych danych (jak to zapisać?) wykonujemy
    - Wczytujemy x, zapamiętujemy x, co z x ujemnymi ?– bo nie ma ujemnych cyfr,
    - Ustawiamy wartość początkową sumy dla wczytanej liczby,
    - Dopóki x jest różne od zera (dlaczego??)
      - „wyciągamy” cyfrę,
      - Zwiększamy sumę,
      - Odrzucamy wyciągnięta i dodaną cyfrę,
    - Poprawiamy max (jak?)
  - Wypisujemy max

# program

```
#include <conio.h>
#include <iostream>
int main()
{
	int i,n,x,max,y,xmax,s,c; //i - która liczba, max - maksimum, x liczba, c - wydobyta cyfra
	cout<<"n=";
	cin>>n;
	max=-1; //moze by max=0, ale max=-1 lepsze, czemu?
	i=1;
	while (i<=n)
	{
		cout<<"x=";
		cin>>x;
		y=x;  //zapamiętujemy x, bo pewnie może się zmieniac
		if (x<0) x=-x;  //rozwiązujemy problem z ujemnymi liczbami, pierwsza zmiana
		s=0;
		while (x!=0)  // można while (x)
		{
			c=x%10; //wyciągamy cyfrę
			s=s+c;
			x=x/10;  // można x/=10   odrzucamy dodaną cyfrę
		}
		if (s>max) //poprawiamy maksimum
		{
			max=s;
			xmax=y;
		}
		i=i+1; //i++
	}
	cout<<"xmax="<<xmax;  //pewien problem jest z przypadkiem, gdy n=0, ile wynosi xmax?
	getch();
	return 0;
}
```

Zwracamy uwagę na komentarze, wcięcia itd..

# Wyniki przykładowego programu



# Omówienie instrukcji

# Lista instrukcji

- Instrukcja pusta
- Instrukcja przypisania
- Instrukcja wywołania funkcji (procedury) – później,
- Instrukcja złożona
- Instrukcje warunkowe
  - if
  - if … else
  - switch
- Instrukcje pętli
  - for
  - while
  - do … while
- Instrukcje skoku:
  - goto,
  - break,
  - continue.

# Instrukcja pusta

- Najprostszą jest **instrukcja pusta**, będąca pustym ciągiem znaków, zakończonym średnikiem:
- `; //instrukcja pusta`
- Instrukcja pusta jest użyteczna w przypadkach gdy składnia wymaga obecności instrukcji, natomiast nie jest wymagane żadne działanie. Nadmiarowe instrukcje puste nie są traktowane przez kompilator jako błędy syntaktyczne, np. zapis
- `int i;`
- składa się z dwóch instrukcji: instrukcji deklaracji int i; oraz instrukcji pustej.
- Jest przydatna także przy tworzeniu złożonych programów.

- Przykłady:
  - i=0; while (i>=0); //petla nieskończona, to zły przykład
  - i=0; while (tablica[i++]>0); //praktyczny przykład, bardzo dobry przykład wykorzystania instrukcji pustej

# Instrukcja złożona

- **Instrukcją złożoną** nazywa się sekwencję instrukcji ujętą w parę nawiasów klamrowych:

```
{
  instrukcja-1;
  instrukcja-2;
  ...
  instrukcja-n;
}
```

# Instrukcja złożona

- W składni języka taka sekwencja jest traktowana jako jedna instrukcja. Instrukcje złożone mogą być zagnieżdżane, np.

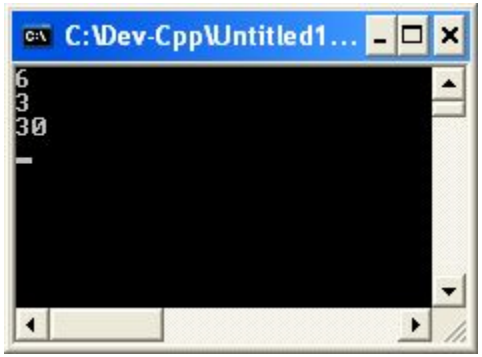
```
{
    instrukcja-1;
    ...
    instrukcja-i;
            {
              instrukcja-j;
              ...
              instrukcja-n;
            }
}
```

- Jeżeli pomiędzy nawiasami klamrowymi występują instrukcje deklaracji identyfikatorów (nazw), to taką instrukcję złożoną nazywamy **blokiem**. Każda nazwa zadeklarowana w bloku ma zasięg od punktu deklaracji do zamykającego blok nawiasu klamrowego. Jeżeli nadamy identyczną nazwę identyfikatorowi, który był już wcześniej zadeklarowany, to poprzednia deklaracja zostanie przesłonięta na czas wykonania danego bloku; po opuszczeniu bloku jej ważność zostanie przywrócona.

# Przykład

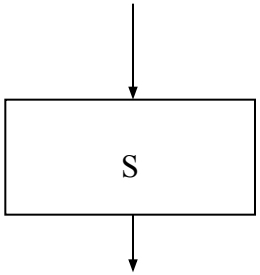
```
#include <conio.h>
#include <iostream.h>
//-----------------------------------------------------------------------------------------------------------
// Program ilustrujący wykorzystanie bloków i zakres działania zmiennych
//-----------------------------------------------------------------------------------------------------------

int main()
{
	int k=0;
	{
		int i=1;
		i=i+2;
		{
			int j=3;
			j=i+3;
			cout<<j<<endl;
			cout<<i<<endl;
		}
		cout<<i;
//		cout<<j;  //źle, dlatego komentarz
	}
	cout<<k<<endl;  //dobrze
//	cout<<i; //źle, dlatego komentarz
//	cout<<j; //źle, dlatego komentarz
	getch();
	return 0;
}
//-----------------------------------------------------------------------------
```



# Instrukcja przypisania

- Dowolne wyrażenie zakończone średnikiem jest nazywane **instrukcją wyrażeniową** (ang. expression statement) lub krótko **instrukcją.** Większość używanych instrukcji stanowią instrukcje-wyrażenia.



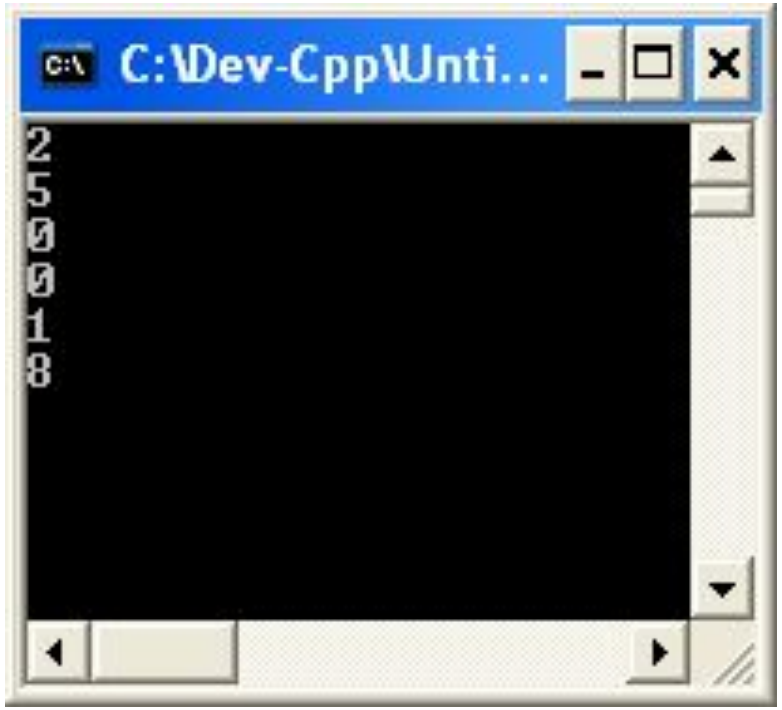

- Jedną z najprostszych jest **instrukcja przypisania** o postaci:
- `zmienna = wyrażenie;`
- gdzie symbol ”=” jest **operatorem przypisania**.

# Instrukcja przypisania

- Przykłady instrukcji języka C++:
  - `int liczba;`
  - `liczba = 2 + 3;`
- Pierwsza z nich jest instrukcją deklaracji; definiuje ona obszar pamięci związany ze zmienną liczba, w którym mogą być przechowywane wartości całkowite. Drugą jest instrukcja przypisania (można powiedzieć, że definiuje zmienną). Umieszcza ona wynik dodawania (wartość wyrażenia) dwóch liczb w obszarze pamięci przeznaczonym na zmienną liczba.
- Można napisać krócej:
  - int liczba=2+3;
- **Przypisanie** jest wyrażeniem, którego wartość jest równa wartości przypisywanej argumentowi z lewej strony operatora przypisania. Instrukcja przypisania powstaje przez dodanie średnika po zapisie przypisania.

# Przykłady instrukcji podstawienia

```
#include <conio.h>
#include <iostream>
//-----------------------------------------------------------------
    -
// Program ilustrujący działanie instrukcji
    podstawienia
//-----------------------------------------------------------------
    -
int main()
{
    int a, b=2, c=3, d=7, e=2, f=5;
    cout <<a<<endl;// A co to???
    a = b + c;
    cout <<a<<endl;
    a = (b + c)/d;
    cout <<a<<endl;
    a = e > f;
    cout <<a<<endl;
    a = (e > f && c < d) + 1;
    cout <<a<<endl;
    a = a << 3;
    cout <<a<<endl;
    getch();
    return 0;
}
```



# Przykład - nieczytelny (?)

- Przykład:

```
a = (b = c + d)/(e = f + g);
```

- jest raczej mało czytelny, równoważny jest sekwencji instrukcji przypisania

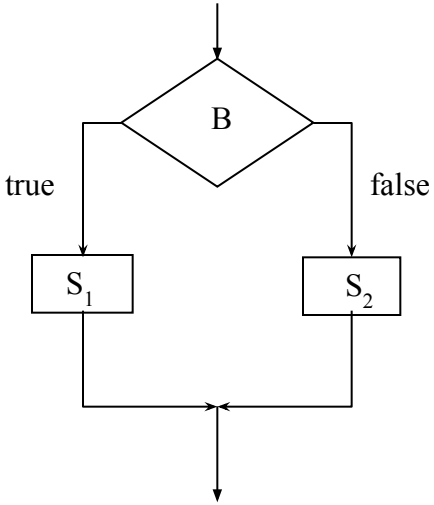
```
b = c + d;
e = f + g;
a = b/e;
```

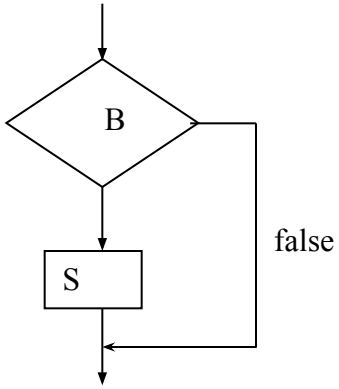
- **Podobnie:**
  - **x1=(-b-sqrt(delta))/(2*a);**
- **Można zapisac jako:**
  - **pom1=-b-sqrt(delta);pom2=2*a;**
  - **x1=pom1/pom2;**
- **To lepszy sposób, niż stosowanie dużej ilości nawiasów.**

# Instrukcje warunkowe

- Instrukcje selekcji (wyboru) wykorzystuje się przy podejmowaniu decyzji.
- Konieczność ich stosowania wynika stąd, że kodowane w języku programowania algorytmy tylko w bardzo prostych przypadkach są czysto sekwencyjne.
- Najczęściej kolejne kroki algorytmu są zależne od spełnienia pewnego warunku lub kilku warunków. Typowe sytuacje - następny slajd.

# Instrukcje warunkowe




- W sytuacji powyższej spełnienie testowanego warunku oznacza wykonanie sekwencji działań $S_1$; przy warunku niespełnionym wykonywana są działania $S_2$ ($S_1$ jest pomijane).
- W sytuacji po prawej stronie pozytywny wynik testu oznacza wykonanie sekwencji działań $S_1$, a wynik negatywny pomija te działania.

# Instrukcja if - składnia

- Instrukcja **if** jest implementacją schematów podejmowania decyzji. Ogólny zapis jest następujący:

  **if (B) instrukcja1; else instrukcja2;**

- lub

  **if (B) instrukcja;**

- gdzie wyrażenie logiczne B musi wystąpić w nawiasach okrągłych.

# Instrukcja if – działanie (semantyka)

- Wykonanie instrukcji **if** zaczyna się od obliczenia wartości wyrażenia B.
- Jeżeli wyrażenie ma wartość różną od zera (prawda!!!), to będzie wykonana instrukcja (lub instrukcja1); jeżeli wyrażenie ma wartość zero (fałsz!!!), to w pierwszym przypadku instrukcja jest wykonywana jest instrukcja2, a w drugim przypadku pomijana.
- Każda z występujących instrukcji może być instrukcją prostą lub złożoną, bądź instrukcją pustą.

# Przydatna uwaga

- `Zapis`
  - `if(wyrażenie != 0)`
- Można zastąpić:
  - `if(wyrażenie)`
- ponieważ test w nawiasach () ma wynik będący wartością numeryczną (liczbową)

# Przykład 1 - najgorsze

```
#include <conio.h>
#include <iostream>
//-----------------------------------------------------------------------
// Program ilustrujący działanie instrukcji if
// Wczytaj trzy liczby, wypisz największą
//-----------------------------------------------------------------------
int main()
{
	int a,b,c;
	cin >>a;
	cin>>b;
	cin>>c;
	if ((a>b) &&(a>c)) cout<<a;
	if ((b>a) &&(b>c)) cout<<b;
	if ((c>b) &&(c>a)) cout<<c;
	getch();
	return 0;
}
//-----------------------------------------------------------------------
//Uwaga: To jest złe rozwiązanie!!. Gdzie jest błąd?
//-----------------------------------------------------------------------
```

# Przykład 2 – lepsze rozwiązanie

```
#include <conio.h>
#include <iostream>
//-----------------------------------------------------------------------------
// Program ilustrujący działanie instrukcji if
// Wczytaj trzy liczby, wypisz największą
//-----------------------------------------------------------------------------
int main()
{
	int a,b,c;
	cin >>a;
	cin>>b;
	cin>>c;
	if ((a>b) && (a>c)) cout<<a;
		else if ((b>a) && (b>c)) cout<<b;
			cout<<c;
	getch();
	return 0;
}
//-----------------------------------------------------------------------------
//Uwaga: To rozwiązanie nie ma błędu
//-----------------------------------------------------------------------------
```

# Uproszczona postać instrukcji if

- Niektóre proste konstrukcje **if** można z powodzeniem zastąpić wyrażeniem warunkowym, wykorzystującym operator warunkowy "?:" . Np.

```
if (a > b)
 max = a;
else
 max = b;
```

- można zastąpić przez

```
max = (a > b) ? a : b;
```

- Nawiasy okrągłe nie są konieczne; dodano je dla lepszej czytelności.

# Przykład

```
#include <conio.h>
#include <iostream.h>
//-----------------------------------------------------------------------------
// Program ilustrujący działanie instrukcji if
// Wczytaj trzy liczby, wypisz największą
//-----------------------------------------------------------------------------
int main()
{
        int a,b,c;
        cin >>a;
        cin>>b;
        cin>>c;
        int d=((a>b)&&(a>c))?a:((b>a)&&(b>c))?b:c;
        cout<<d;
        getch();
        return 0;
}
//-----------------------------------------------------------------------------
```

# Instrukcja switch -wprowadzenie

- Rozważmy problem wypisania liczby dni w miesiącu dla zwykłego roku:
- Będzie ona postaci:

```
cin>>nr_m;
if (nr_m==1) cout <<31;
else if nr_m==2 cout<<28;
        else if nr_m==3 cout<<31;
              else if nr_m==4 cout<<30;
                  else if nr_m==5 cout<<31;
                       else if nr_m==6 cout<<30;
                             else if nr_m==7 cout<<31;
                            else if nr_m==8 cout<<31;
                                 else if nr_m==9 cout<<30;
                                        else if nr_m==10 cout<<31;
                                 else if nr_m==11 cout<<30;
                                         else cout<<31;
```

- Jakie są wady takiego tekstu programu?

# Instrukcja switch

- Instrukcja **switch** służy do podejmowania decyzji wielowariantowych, gdy zastosowanie instrukcji **if-else** prowadziłoby do zbyt głębokich zagnieżdżeń i ewentualnych niejednoznaczności. Składnia instrukcji jest następująca:
- `switch (wyrażenie) instrukcja;`
- gdzie instrukcja jest zwykle instrukcją złożoną (blokiem), której instrukcje składowe są poprzedzane słowem kluczowym **case** z etykietą; wyrażenie, które musi przyjmować wartości całkowite, spełnia tutaj rolę *selektora wyboru*.

# Instrukcja switch

- W rozwiniętym zapisie

```
switch (wyrażenie)
{
        case etykieta_1 : instrukcje;
           ...
        case etykieta_n : instrukcje;
        default         : instrukcje;
}
```

# Instrukcja switch

- Etykiety są całkowitymi wartościami stałymi lub wyrażeniami stałymi. Nie może być dwóch identycznych etykiet. Jeżeli jedna z etykiet ma wartość równą wartości wyrażenia **wyrażenie**, to wykonanie zaczyna się od tego przypadku (ang. case - przypadek) i przebiega aż do końca bloku. Instrukcje po etykiecie `default` są wykonywane wtedy, gdy żadna z poprzednich etykiet nie ma aktualnej wartości selektora wyboru.
- Przypadek z etykietą `default` jest opcją: jeżeli nie występuje i żadna z pozostałych etykiet nie przyjmuje wartości selektora, to nie jest podejmowane żadne działanie i sterowanie przechodzi do następnej po **switch** instrukcji programu.

# Instrukcja switch - komentarz

- Z opisu wynika, że instrukcja **switch** jest co najwyżej dwualternatywna i mówi: wykonuj wszystkie instrukcje od danej etykiety do końca bloku, albo: wykonuj instrukcje po default do końca bloku (bądź żadnej, jeśli default nie występuje).
- Wersja ta nie wychodzi zatem poza możliwości instrukcji **if**, bądź **if-else**.
- Dla uzyskania wersji z wieloma wzajemnie wykluczającymi się alternatywami musimy odseparować poszczególne przypadki w taki sposób, aby po wykonaniu instrukcji dla wybranego przypadku sterowanie opuściło blok instrukcji **switch**. Możliwość taką zapewnia instrukcja **break**.

# Instrukcja switch

- Zatem dla selekcji wyłącznie jednego z wielu wariantów składnia instrukcji **switch** będzie miała postać:

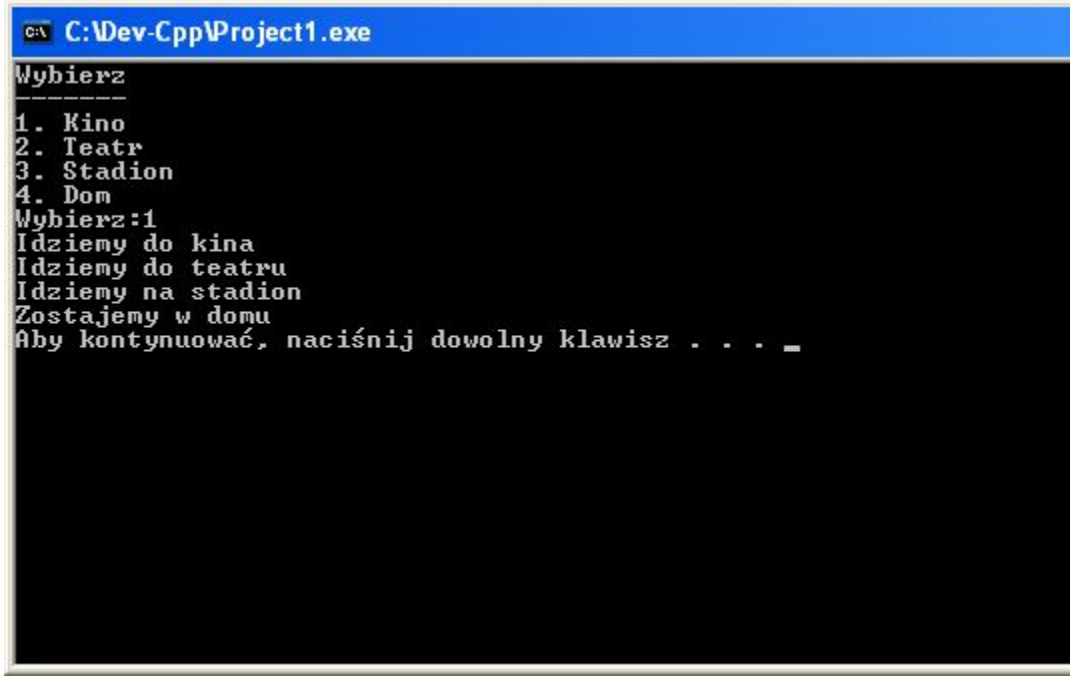
```
switch(wyrażenie)
{
        case etykieta_1 : instrukcje;
                          break;
            ...
        case etykieta_n : instrukcje;
                          break;
        default   : instrukcje;
                    break;
}
```

# Instrukcja switch – przykład źle zaprogramowany

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    cout<<"Wybierz\n";
    cout<<"-------\n";
    cout<<"1. Kino\n";
    cout<<"2. Teatr\n";
    cout<<"3. Stadion\n";
    cout<<"4. Dom\n";
    cout<<"Wybierz:";
    int wybor;
    cin>>wybor;
    switch(wybor)
    {
      case 1:
          cout<<"Idziemy do kina"<<endl;
      case 2:
          cout<<"Idziemy do teatru"<<endl;
      case 3:
          cout<<"Idziemy na stadion"<<endl;
      case 4:
          cout<<"Zostajemy w domu"<<endl;
    }
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

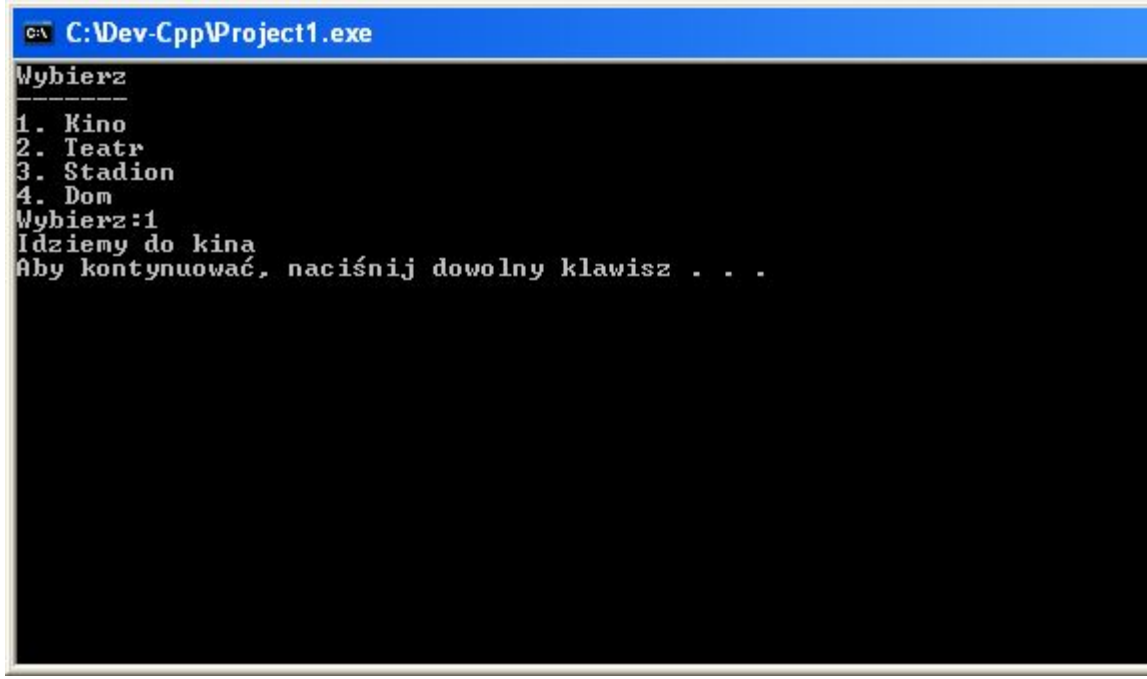


# Instrukcja switch – przykład poprawiony

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
   cout<<"Wybierz\n";
   cout<<"-------\n";
   cout<<"1. Kino\n";
   cout<<"2. Teatr\n";
   cout<<"3. Stadion\n";
   cout<<"4. Dom\n";
   cout<<"Wybierz:";
   int wybor;
   cin>>wybor;
   switch(wybor)
   {
     case 1:
         cout<<"Idziemy do kina"<<endl;
         break;
     case 2:
         cout<<"Idziemy do teatru"<<endl;
         break;
     case 3:
         cout<<"Idziemy na stadion"<<endl;
         break;
     case 4:
         cout<<"Zostajemy w domu"<<endl;
         break;
     default: cout<<"Zły wybor";
   }
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

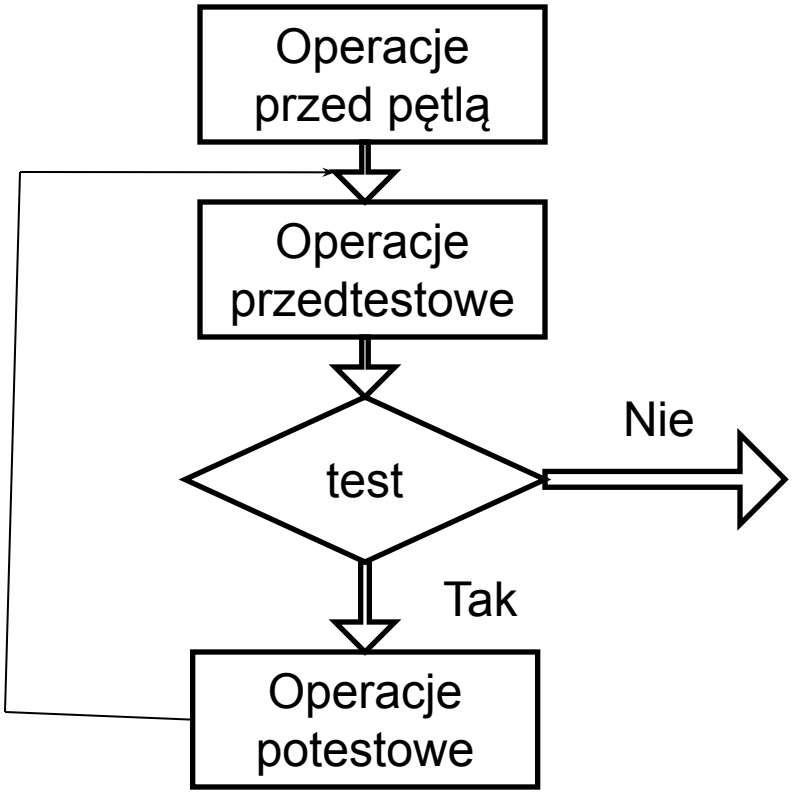


# Instrukcje iteracyjne

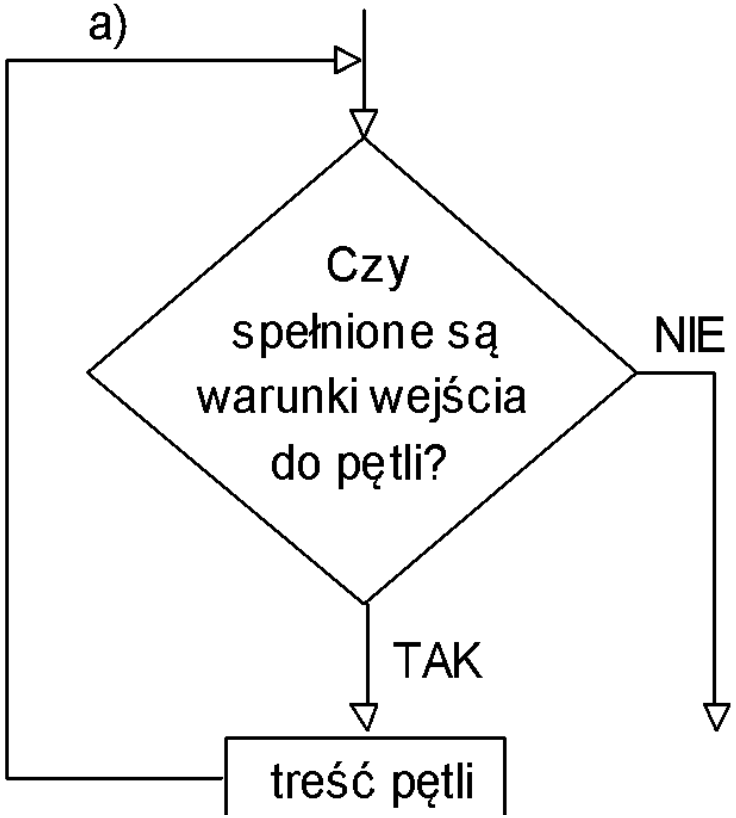
- Instrukcje powtarzania (pętli) lub iteracji pozwalają wykonywać daną instrukcję, prostą lub złożoną, zero lub więcej razy. W języku C++ mamy do dyspozycji trzy instrukcje iteracyjne (pętli):
  - **while**,
  - **do-while**
  - **for**.
- Różnią się one przede wszystkim metodą sterowania pętlą. Dla pętli **while** i **for** sprawdza się warunek wejścia do pętli, natomiast dla pętli **do-while** sprawdza się warunek wyjścia z pętli.

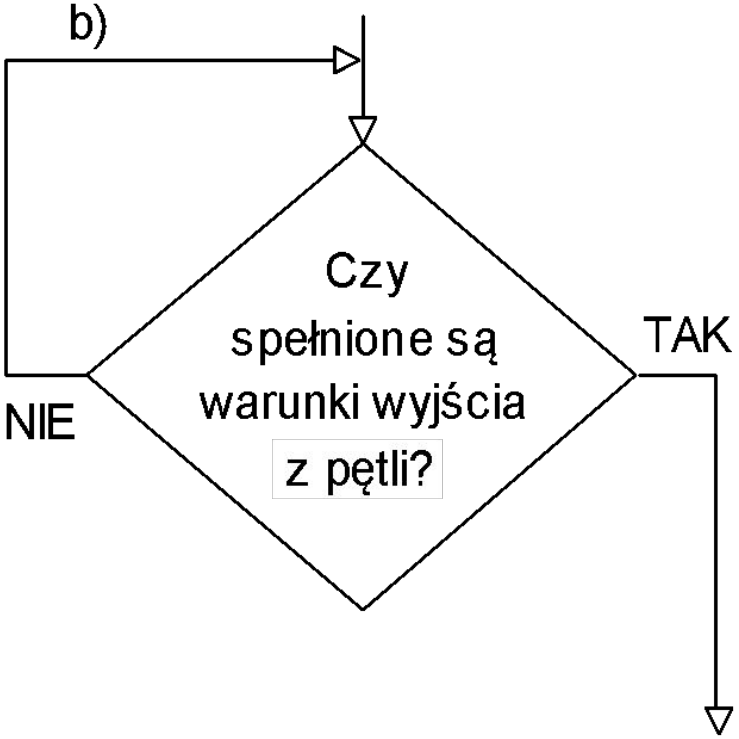
# Bardzo ogólna budowa pętli

- Nie występuje w żadnym języku programowania,



# Schemat działania pętli - praktyczne



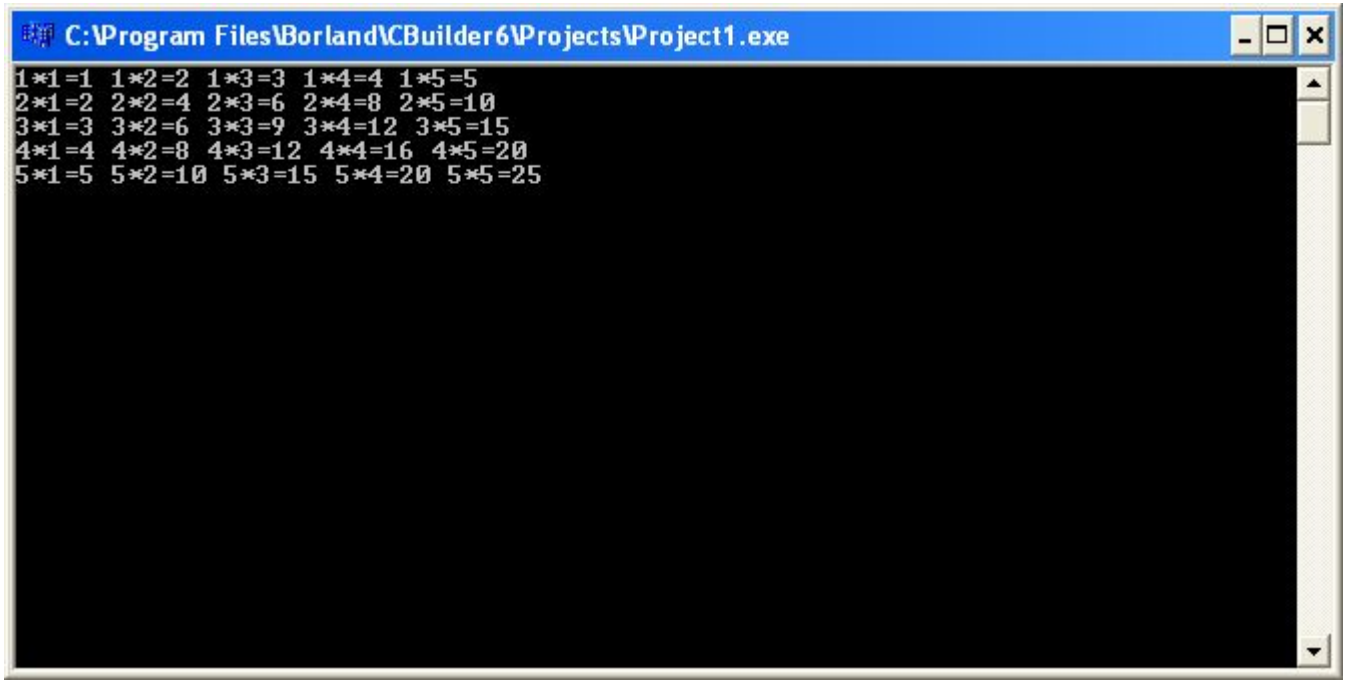
# Zagnieżdżanie instrukcji pętli

- Instrukcje iteracyjne, podobnie jak instrukcje selekcji i warunkowe, można zagnieżdżać do dowolnego poziomu zagnieżdżenia.
- Jednak przy wielu poziomach zagnieżdżenia program staje się mało czytelny.
- W praktyce nie zaleca się stosować więcej niż trzech poziomów zagnieżdżenia.

# Przykład zagnieżdżenia

```
#include <iostream.h>
#include <conio.h>
//-----------------------------------------------------------------------
int main()
{
        //tabliczka mnożenia
        for (int i=1;i<=5;i++)
        {
             for (int j=1;j<=5;j++)
                cout<<i<<'*'<<j<<"="<<i*j<<" ";
             cout<<endl;
        }
        getch();
        return 0;
}
//-----------------------------------------------------------------------
```

# Wynik działania



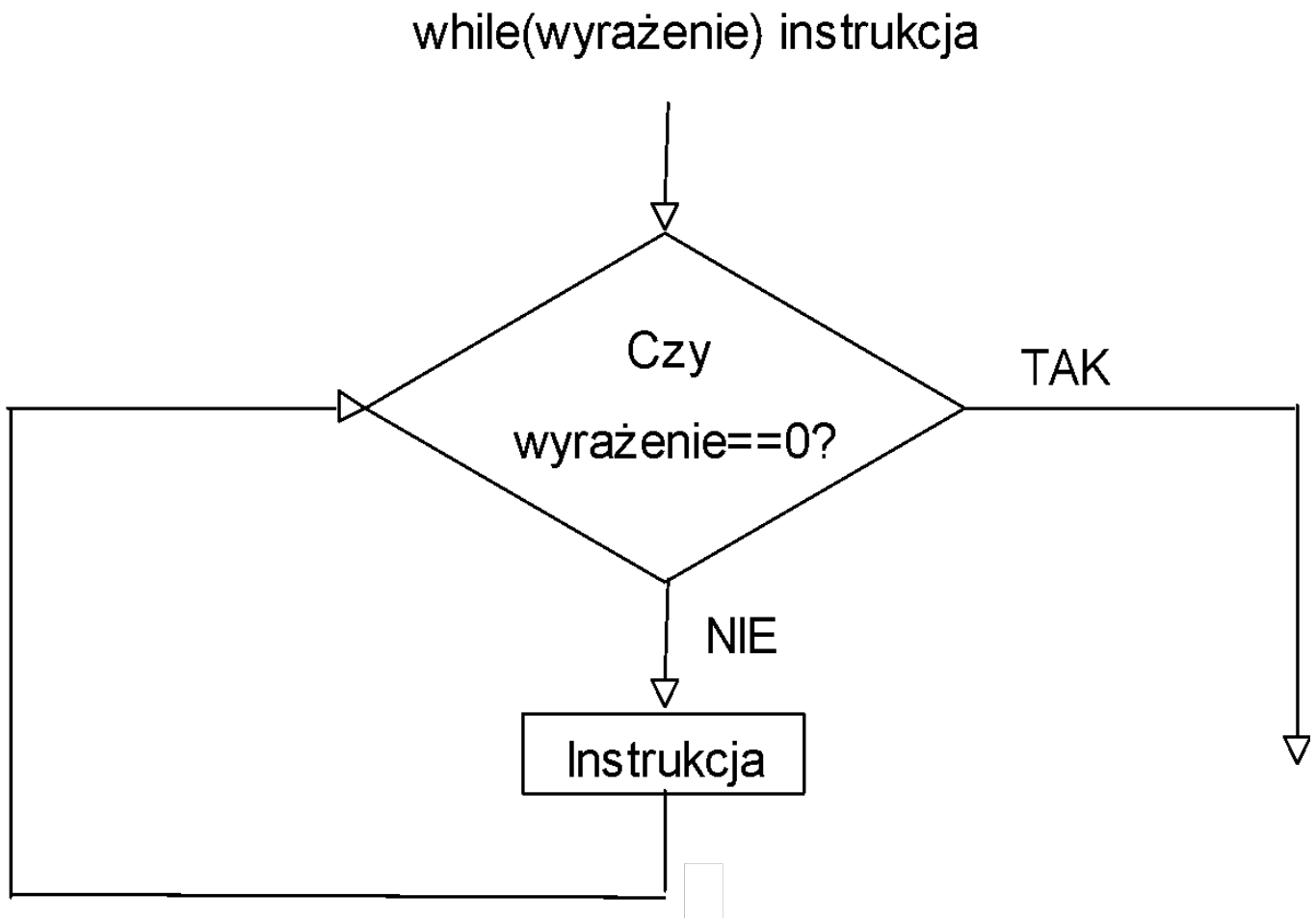
# Instrukcja while

- Składnia instrukcji **while** jest następująca:

```
while (wyrażenie) instrukcja;
```

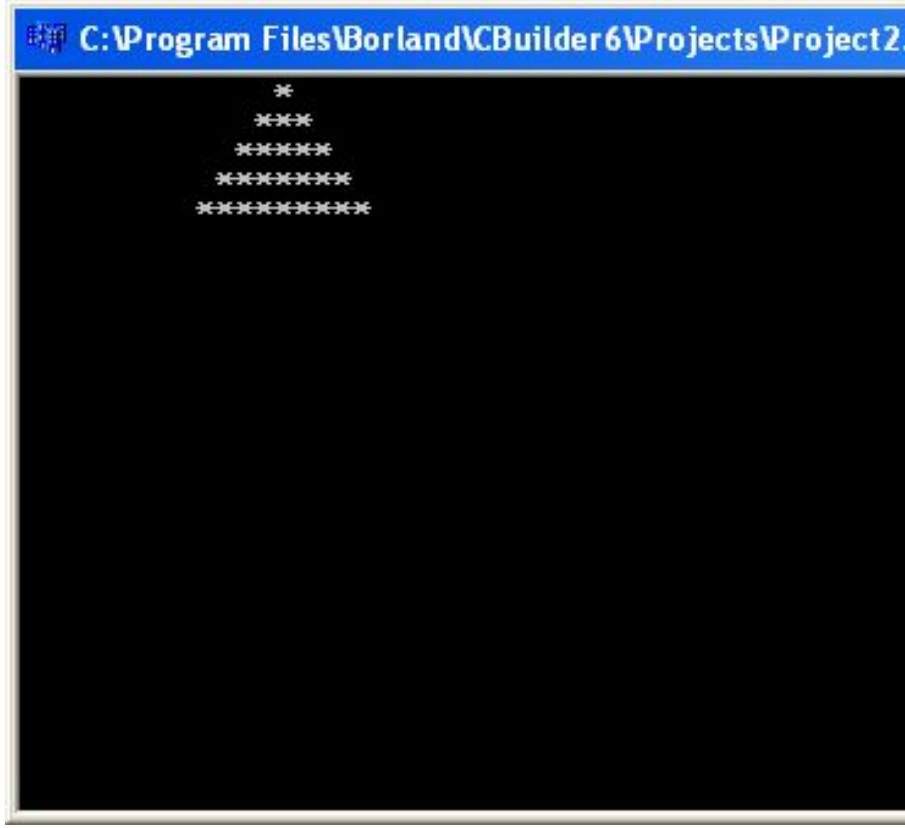
- gdzie `instrukcja` może być instrukcją pustą, instrukcją prostą lub instrukcją złożoną.
- Sekwencja działań przy wykonywaniu instrukcji **while** jest następująca:
    1. Oblicz wartość wyrażenia i sprawdź, czy jest równe zeru (fałsz). Jeżeli tak, to pomiń krok 2; jeżeli nie (prawda), przejdź do kroku 2.
    2. Wykonaj instrukcję i przejdź do kroku 1.
- Jeżeli pierwsze wartościowanie wyrażenia wykaże, że ma ono wartość zero, to instrukcja nie zostanie wykonana i sterowanie przejdzie do następnej instrukcji programu.

# Schemat blokowy dla while



# Przykład

```
#include <conio.h>
#include <iostream.h>
#include <iomanip.h>
//-----------------------------------------------------------------------------
// Program ilustrujący działanie instrukcji while
// Funkcja setw(liczba) powoduje ustawienie kursora w kolumnie o
// podanym numerze i bierzącym wierszu.
//-----------------------------------------------------------------------------
int main() {
   const int WIERSZ = 5;
   const int KOLUMNA = 15;
   int j, i = 1;
   while(i <= WIERSZ)
    {
      cout << setw(KOLUMNA - i) << '*';
      j = 1;
      while( j <= 2*i-2 )
        {
         cout << '*';
         j++;
        }
      cout << endl;
      i++;
    }
    getch();
   return 0;
}
```



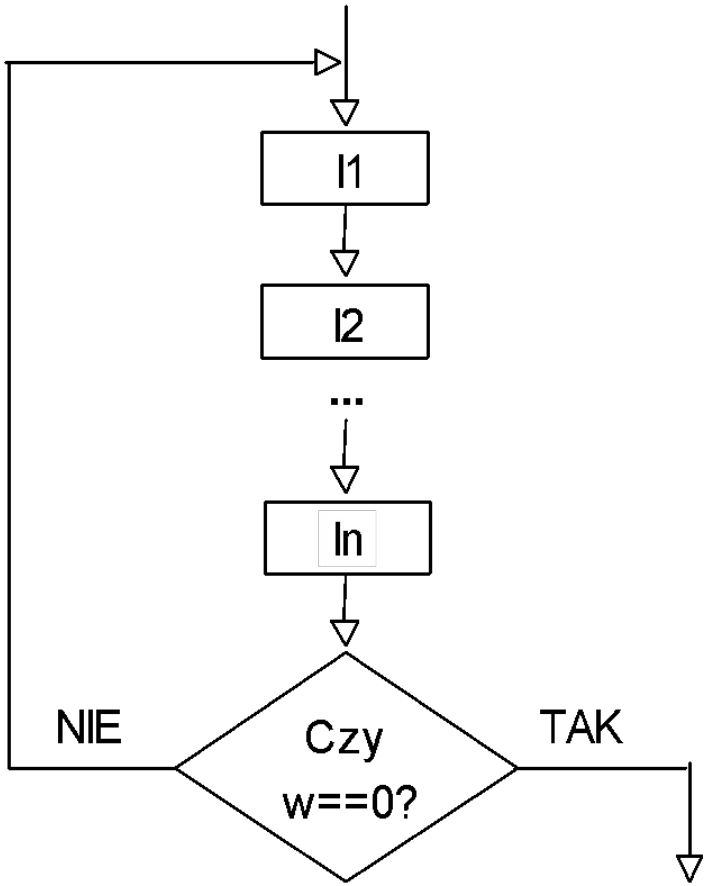
# Komentarz

- W programie mamy zagnieżdżoną pętlę **while**. Pierwsze sprawdzenie wyrażenia j <= 2*i-2 w pętli wewnętrznej daje wartość 0 (fałsz), zatem w pierwszym obiegu pętla wewnętrzna zostaje pominięta.
- W drugim sprawdzeniu pętla wewnętrzna dorysowuje dwie gwiazdki, itd., aż do wyjścia z pętli zewnętrznej.
- Nowym elementem programu jest włączenie pliku nagłówkowego iomanip.h, w którym znajduje się deklaracja funkcji setw(int w). Funkcja ta służy do ustawiania miejsca wpisywania znaków (kolumny) w bieżącym wierszu.

# Instrukcja do-while

- Składnia instrukcji **do-while** ma postać:
- `do instrukcja; while (wyrażenie)`
- gdzie `instrukcja` może być instrukcją pustą, instrukcją prostą lub złożoną.
- W pętli *do-while* instrukcja (ciało pętli) zawsze będzie wykonana co najmniej jeden raz, ponieważ test na równość zeru wyrażenia jest przeprowadzany po jej wykonaniu. Pętla kończy się po stwierdzeniu, że wyrażenie jest równe zeru.

# Schemat blokowy do-while




```
do {
      I1
      I2
      ...
      In }

while(w);
```

I1, I2, ...,In - instrukcje

w - wyrażenie

# Przykład

```
#include <conio.h>
#include <iostream.h>

int main() {
    char znak;
    cout << "Wprowadz dowolny znak;\n* oznacza koniec.\n";
    do {
        cout << ": ";
        cin  >> znak;
     }
   while  (znak !='*');
   return 0;
}
```

# Instrukcja for

- Jest to, podobnie jak instrukcja **while**, instrukcja sterująca powtarzaniem ze sprawdzeniem warunku zatrzymania na początku pętli. Stosuje się ją w tych przypadkach, gdy znana jest liczba obiegów pętli. Składnia instrukcji **for** jest następująca:
  - `for(instrukcja-inicjująca ;wyrażenie1;wyrażenie2) instrukcja;`
- gdzie instrukcja może być instrukcją pustą, instrukcją prostą lub złożoną.

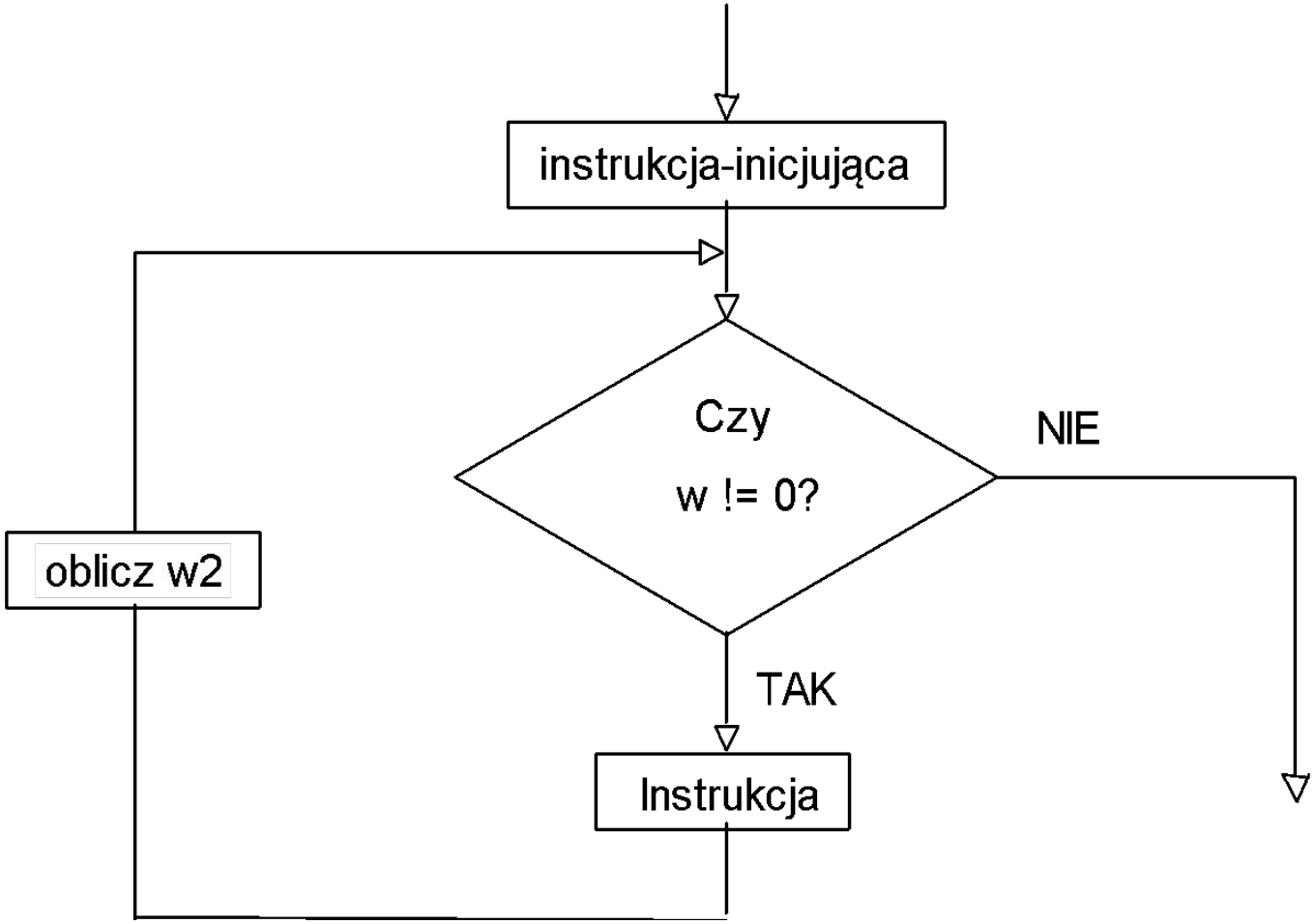
# Instrukcja for

- Algorytm obliczeń dla pętli *for* jest następujący:

1. Wykonaj instrukcję o nazwie *instrukcja-inicjująca*. Zwykle będzie to zainicjowanie jednego lub kilku liczników pętli (zmiennych sterujących), ewentualnie inicjująca instrukcja deklaracji, np.
   - `for (i = 0; ...`
   - `for (i =0, j = 1; ...`
   - `for (int i = 0; ...`
   - Instrukcja inicjująca może być również instrukcją pustą, jeżeli zmienna sterująca została już wcześniej zadeklarowana i zainicjowana, np.
   - `int i = 1; for (; ...`
2. Oblicz wartość wyrażenia *wyrażenie1* i porównaj ją z zerem. Jeżeli *wyrażenie1* ma wartość różną od zera (prawda) przejdź do kroku 3. Jeżeli *wyrażenie1* ma wartość zero, opuść pętlę.
3. Wykonaj instrukcję *instrukcja* i przejdź do kroku 4.
4. Oblicz wartość wyrażenia *wyrażenie2* (zwykle oznacza to zwiększenie licznika pętli) i przejdź do kroku 2.

# Schemat blokowy instrukcji for

for(instrukcja-inicjująca;w;w2) Instrukcja;



# Pętla for a while

- Instrukcja **for** jest równoważna następującej instrukcji **while**:

```
instrukcja-inicjująca;
while (w)
{
 instrukcja;
 w2;  }
```

# Złożone sytuacje dla for

- Ważnym elementem składni instrukcji **for** jest sposób zapisu instrukcji inicjującej oraz wyrażeń składowych, gdy mamy kilka zmiennych sterujących. W takich przypadkach przecinek pomiędzy wyrażeniami pełni rolę operatora. Np. w instrukcji
  - for ( i = 1, j = 2; i< 5; i++, j++ ) cout <<"i =" << i << " j = " << j << "\n";
- instrukcja inicjująca zawiera dwa wyrażenia: `i = 1` oraz `j = 2` połączone przecinkiem. Operator przecinkowy ',' wiąże te dwa wyrażenia w jedno wyrażenie, wymuszając wartościowanie wyrażeń od lewej do prawej. Tak więc najpierw `ii` zostaje zainicjowane do 1, a następnie `jj` zostanie zainicjowane do 2. Podobnie wyrażenie2, które składa się z dwóch wyrażeń `i++` oraz `j++`, połączonych operatorem przecinkowym; po każdym wykonaniu instrukcji `cout` najpierw zwiększa się o 1 `i`, a następnie `j`.

# Przykład

```
#include <conio.h>
#include <iostream.h>
#include <iomanip.h>
//Program Piramida
//zamiast while mamy for
int main()  {
   const int WIERSZ = 5;
   const int KOLUMNA = 15;
   for (int i = 1; i <= WIERSZ; i++)
     {
      cout << setw(KOLUMNA - i) << '*';
      for (int j = 1; j <= 2 * i -2; j++)
      cout << '*';
      cout << endl;
     }
    getch();
   return 0;
}
```

# Omówienie przykładu

- W przykładzie, podobnie jak dla instrukcji **while**, wykorzystano funkcję `setw()` z pliku `iomanip.h`. Identyczne są również definicje stałych symbolicznych `WIERSZ` i `KOLUMNA`.
- Taka sama jest również postać wydruku. Natomiast program jest nieco krótszy; wynika to stąd, że instrukcja **for** jest wygodniejsza od instrukcji **while** dla znanej z góry liczby obiegów pętli.
- Zauważmy też, że w pętli wewnętrznej umieszczono definicję zmiennej `j` typu **int**. Zasięg tej zmiennej jest ograniczony: można się nią posługiwać tylko od punktu definicji do końca bloku zawierającego wewnętrzną instrukcję **for**.

# Szczególny przypadek

- Syntaktycznie poprawny jest zapis

  ```
  for (; ;)
  ```

- Jest to zdegenerowana postać instrukcji **for**, równoważna

  ```
  for(;1;)
  ```

- lub

  ```
  while (1)
  ```

- czyli pętli nieskończonej; np. instrukcja

  ```
  for(;;) cout << "wiersz\n";
  ```

- będzie drukować podany tekst aż do zatrzymania programu, lub wymuszenia instrukcją **break** zatrzymania pętli.

# Podpowiedzi dotyczące stosowania pętli

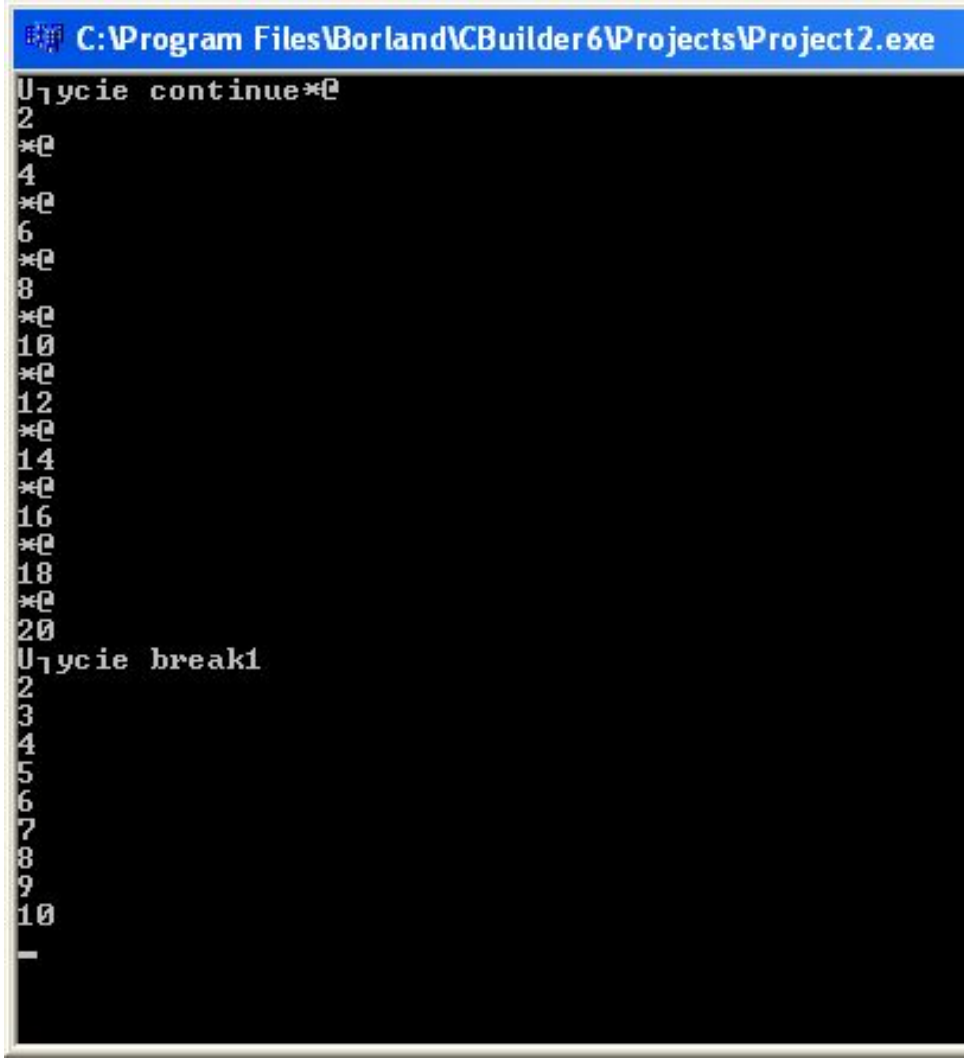
1. Jeśli wiemy, ile razy pętla się wykona, to używajmy pętli for, może wykonać się 5 razy, n razy, $n^2$ razy, f(n) razy,
2. Jeśli nie wiemy ile razy (np. wprowadzaj dane, aż napotkasz ”coś”) to uzywamy petli:
   1. while, gdy warunek testujemy na początku
   2. do .. while, gdy warunek testujemy na końcu
3. Każda pętla jest równoważna innej w tym sensie, że każdy algorytm zapisany za pomocą jednej pętli może być zapisany za pomocą innej. Być może będą potrzebne dodatkowe instrukcje, ale jest to zawsze możliwe.

# Inne instrukcje

- W ciele instrukcji iteracyjnych używa się niekiedy instrukcji **continue;**. Wykonanie tej instrukcji przekazuje sterowanie do części testującej wartość wyrażenia w pętlach **while** i **do-while** (krok 1), lub do kroku 4 w instrukcji **for**.
- Instrukcję iteracyjną można przerwać za pomocą break;.
- W języku C++ istnieje instrukcja skoku bezwarunkowego **goto** o składni goto etykieta; Jej używanie nie jest zalecane.

# Przykład dla brak i continue

```
#include <conio.h>
#include <iostream.h>
#include <iomanip.h>
//instrukcja continue i  break;
int main()  {
    cout<<"Użycie continue";
    for (int i = 1; i <= 20; i++)
    {
        if (i%2==0)
        {
                cout<<i<<endl;
                continue;
        } else cout<<'*';
        cout<<'@‘<<endl;
    }
    cout<<"Użycie break";
    for (int i = 1; i <= 20; i++)
        if (i>10) break;
        else cout<<i<<endl;
     getch();
    return 0;
}
```



# Przykład- użycie instrukcji skoku

```
#include <conio.h>
#include <iostream.h>
#include <iomanip.h>
//instrukcja skoku - np. zamiast while;
//sumuj 10 liczb całkowitych
int main()
{
	int i=1,s=0,x;
	powrot: if (i>10) goto et;
	cin>>x;
	s=s+x;
	i++;
	goto powrot;
	et:	cout<<s;
	getch();
	return 0;
}
```